FOUNDATION-0.3: ClickHouse Time-Series Schema - PART 1 (Code)

🎯 CONTEXT

Phase: FOUNDATION (Week 1 - Day 2 Evening)

Component: ClickHouse Time-Series Database Setup

Estimated Time: 15 min AI execution + 10 min verification

Complexity: MEDIUM

Risk Level: LOW

Files: Part 1 of 2 (Code implementation)

MILESTONE: High-frequency metrics storage for all agents! 📊

📦 DEPENDENCIES

Must Complete First:

- **P-01:** Bootstrap Project Structure ✅ COMPLETED
- **FOUNDATION-0.2a:** Core Database Schema ✅ COMPLETED
- **FOUNDATION-0.2b:** Agent State Tables ✅ COMPLETED
- **FOUNDATION-0.2c:** Workflow History Tables ✅ COMPLETED
- **FOUNDATION-0.2d:** Resource Schema Tables ✅ COMPLETED
- **FOUNDATION-0.2e:** Application Schema Tables ✅ COMPLETED

Required Services Running:

```bash
# Verify all services are healthy
cd ~/optiinfra
make verify

# Expected output:
# PostgreSQL... ✅ HEALTHY
# ClickHouse... ✅ HEALTHY
# Qdrant... ✅ HEALTHY
# Redis... ✅ HEALTHY
```

Verify ClickHouse is Accessible:

```bash
# Test ClickHouse connection
docker exec optiinfra-clickhouse clickhouse-client --query="SELECT 1"

# Expected output: 1
```

🎯 OBJECTIVE

Set up **ClickHouse time-series database** to store high-frequency metrics that would overwhelm PostgreSQL.

What We're Building:

4 Time-Series Tables (raw metrics):

1. **cost_metrics_ts** - Cloud costs per instance per hour (90-day TTL)
2. **performance_metrics_ts** - LLM latency/throughput per request (30-day TTL)
3. **resource_metrics_ts** - GPU/CPU utilization per minute (90-day TTL)
4. **quality_metrics_ts** - Quality scores per request (30-day TTL)

4 Materialized Views (hourly aggregations):

1. **cost_metrics_hourly_mv** - Pre-aggregated hourly costs
2. **performance_metrics_hourly_mv** - P95/P99 latency by hour
3. **resource_metrics_hourly_mv** - GPU utilization by hour
4. **quality_metrics_hourly_mv** - Quality trends by hour

Python Client - Easy-to-use interface for inserts and queries

Database Architecture:

```
┌──────────────────────────────────────────────────┐
│      POSTGRESQL (0.2a-0.2e)                      │
│  18 tables, 21 enums, 115+ indexes               │
│     (Structured transactional)                   │
└──────────────────────┬───────────────────────────┘
                       │
                       │ Customer/Agent UUIDs
```

```
                    ▼
┌──────────────────────────────────────────────────────┐
│      CLICKHOUSE (0.3) ✅ NEW                         │
│   8 tables (4 base + 4 views)                        │
│   (High-frequency time-series)                       │
│                                                      │
│  Performance:                                        │
│  - 1M+ inserts/second                                │
│  - Millisecond query latency                         │
│  - 10-20x compression                                │
│  - Automatic TTL cleanup                             │
└──────────────────────────────────────────────────────┘
```

Why ClickHouse?

Feature	PostgreSQL	ClickHouse
Time-series inserts	~10K/sec	~1M/sec (100x faster)
Query latency	Seconds	Milliseconds
Storage compression	2-3x	10-20x
Best for	Structured data	Time-series metrics

📁 FILE 1: ClickHouse Initialization Script

Location: `~/optiinfra/shared/clickhouse/migrations/init.sql`

sql

```
-- ============================================================================
-- OptiInfra ClickHouse Time-Series Database
-- Foundation Phase 0.3
-- ============================================================================

-- Create database
CREATE DATABASE IF NOT EXISTS optiinfra;
USE optiinfra;

-- ============================================================================
-- TABLE 1: cost_metrics_ts
-- Purpose: Track cloud costs per instance per hour
-- Retention: 90 days
-- Granularity: 1 hour
-- ============================================================================

CREATE TABLE IF NOT EXISTS cost_metrics_ts (
    timestamp DateTime,
    customer_id UUID,
    cloud_provider String,        -- 'aws', 'gcp', 'azure'
    service_name String,          -- 'ec2', 'compute-engine', 'vm'
    instance_id String,
    instance_type String,         -- 'm5.xlarge', 'n1-standard-4'
    region String,                -- 'us-east-1', 'us-central1'
    cost_per_hour Float64,
    utilization_percent Float32,
    is_spot UInt8,                -- 0 or 1 (boolean)
    is_reserved UInt8             -- 0 or 1 (boolean)
)
ENGINE = MergeTree()
PARTITION BY toYYYYMM(timestamp)
ORDER BY (customer_id, cloud_provider, timestamp)
TTL timestamp + INTERVAL 90 DAY
SETTINGS index_granularity = 8192;

-- ============================================================================
-- TABLE 2: performance_metrics_ts
-- Purpose: Track LLM inference performance per request
-- Retention: 30 days
-- Granularity: 1 second (per request)
-- ============================================================================

CREATE TABLE IF NOT EXISTS performance_metrics_ts (
```

```
    timestamp DateTime,
    customer_id UUID,
    service_id UUID,
    service_type String,              -- 'vllm', 'tgi', 'sglang'
    model_name String,                -- 'gpt-4', 'llama-2-70b'
    request_id UUID,
    latency_ms Float32,
    throughput_tokens_per_sec Float32,
    gpu_utilization Float32,
    kv_cache_utilization Float32,
    batch_size UInt32,
    prompt_tokens UInt32,
    completion_tokens UInt32,
    total_tokens UInt32
)
ENGINE = MergeTree()
PARTITION BY toYYYYMM(timestamp)
ORDER BY (customer_id, service_id, timestamp)
TTL timestamp + INTERVAL 30 DAY
SETTINGS index_granularity = 8192;

-- ============================================================================
-- TABLE 3: resource_metrics_ts
-- Purpose: Track GPU/CPU utilization per minute
-- Retention: 90 days
-- Granularity: 1 minute
-- ============================================================================

CREATE TABLE IF NOT EXISTS resource_metrics_ts (
    timestamp DateTime,
    customer_id UUID,
    instance_id String,
    instance_type String,
    gpu_index UInt8,                  -- 0-7 (which GPU on instance)
    gpu_utilization Float32,          -- 0-100%
    gpu_memory_used_mb Float32,
    gpu_memory_total_mb Float32,
    gpu_temperature Float32,
    cpu_utilization Float32,          -- 0-100%
    memory_used_gb Float32,
    memory_total_gb Float32,
    network_rx_mbps Float32,
    network_tx_mbps Float32
)
```

```
ENGINE = MergeTree()
PARTITION BY toYYYYMM(timestamp)
ORDER BY (customer_id, instance_id, timestamp)
TTL timestamp + INTERVAL 90 DAY
SETTINGS index_granularity = 8192;

-- ============================================================================
-- TABLE 4: quality_metrics_ts
-- Purpose: Track LLM output quality per request
-- Retention: 30 days
-- Granularity: Per request
-- ============================================================================

CREATE TABLE IF NOT EXISTS quality_metrics_ts (
    timestamp DateTime,
    customer_id UUID,
    service_id UUID,
    request_id UUID,
    model_name String,
    relevance_score Float32,          -- 0-1
    coherence_score Float32,          -- 0-1
    factuality_score Float32,         -- 0-1
    hallucination_detected UInt8,     -- 0 or 1
    toxicity_score Float32,           -- 0-1
    overall_quality_score Float32,    -- weighted average
    prompt_hash String,               -- for grouping similar prompts
    latency_ms UInt32
)
ENGINE = MergeTree()
PARTITION BY toYYYYMM(timestamp)
ORDER BY (customer_id, service_id, timestamp)
TTL timestamp + INTERVAL 30 DAY
SETTINGS index_granularity = 8192;

-- ============================================================================
-- MATERIALIZED VIEW 1: cost_metrics_hourly_mv
-- Purpose: Pre-aggregate cost data by hour for fast dashboard queries
-- ============================================================================

CREATE MATERIALIZED VIEW IF NOT EXISTS cost_metrics_hourly_mv
ENGINE = SummingMergeTree()
PARTITION BY toYYYYMM(hour)
ORDER BY (customer_id, cloud_provider, hour)
POPULATE
```

```
AS SELECT
    toStartOfHour(timestamp) as hour,
    customer_id,
    cloud_provider,
    service_name,
    sum(cost_per_hour) as total_cost,
    avg(utilization_percent) as avg_utilization,
    count() as sample_count
FROM cost_metrics_ts
GROUP BY hour, customer_id, cloud_provider, service_name;

-- ============================================================================
-- MATERIALIZED VIEW 2: performance_metrics_hourly_mv
-- Purpose: Pre-aggregate latency P95/P99 by hour
-- ============================================================================

CREATE MATERIALIZED VIEW IF NOT EXISTS performance_metrics_hourly_mv
ENGINE = AggregatingMergeTree()
PARTITION BY toYYYYMM(hour)
ORDER BY (customer_id, service_id, hour)
POPULATE
AS SELECT
    toStartOfHour(timestamp) as hour,
    customer_id,
    service_id,
    service_type,
    avgState(latency_ms) as avg_latency,
    quantileState(0.95)(latency_ms) as p95_latency,
    quantileState(0.99)(latency_ms) as p99_latency,
    avgState(throughput_tokens_per_sec) as avg_throughput,
    avgState(gpu_utilization) as avg_gpu_util,
    count() as request_count
FROM performance_metrics_ts
GROUP BY hour, customer_id, service_id, service_type;

-- ============================================================================
-- MATERIALIZED VIEW 3: resource_metrics_hourly_mv
-- Purpose: Pre-aggregate GPU utilization by hour
-- ============================================================================

CREATE MATERIALIZED VIEW IF NOT EXISTS resource_metrics_hourly_mv
ENGINE = AggregatingMergeTree()
PARTITION BY toYYYYMM(hour)
ORDER BY (customer_id, instance_id, hour)
```

```
POPULATE
AS SELECT
    toStartOfHour(timestamp) as hour,
    customer_id,
    instance_id,
    instance_type,
    avgState(gpu_utilization) as avg_gpu_util,
    maxState(gpu_utilization) as max_gpu_util,
    avgState(gpu_memory_used_mb) as avg_gpu_memory,
    avgState(cpu_utilization) as avg_cpu_util,
    count() as sample_count
FROM resource_metrics_ts
GROUP BY hour, customer_id, instance_id, instance_type;

-- ============================================================================
-- MATERIALIZED VIEW 4: quality_metrics_hourly_mv
-- Purpose: Pre-aggregate quality scores by hour
-- ============================================================================

CREATE MATERIALIZED VIEW IF NOT EXISTS quality_metrics_hourly_mv
ENGINE = AggregatingMergeTree()
PARTITION BY toYYYYMM(hour)
ORDER BY (customer_id, service_id, hour)
POPULATE
AS SELECT
    toStartOfHour(timestamp) as hour,
    customer_id,
    service_id,
    model_name,
    avgState(overall_quality_score) as avg_quality,
    avgState(relevance_score) as avg_relevance,
    avgState(coherence_score) as avg_coherence,
    avgState(factuality_score) as avg_factuality,
    sumState(hallucination_detected) as hallucination_count,
    count() as request_count
FROM quality_metrics_ts
GROUP BY hour, customer_id, service_id, model_name;

-- ============================================================================
-- VERIFICATION QUERIES
-- ============================================================================

-- Show all tables
SHOW TABLES;
```

```
-- Check table row counts
SELECT
    database,
    table,
    formatReadableSize(total_bytes) as size,
    total_rows as rows
FROM system.tables
WHERE database = 'optiinfra'
ORDER BY table;
```

📁 FILE 2: Python ClickHouse Client

Location: `~/optiinfra/shared/clickhouse/client.py`

```
python
```

```
"""
ClickHouse client for high-frequency time-series metrics.
Provides easy-to-use interface for inserting and querying metrics.

Usage:
    from shared.clickhouse.client import get_clickhouse_client

    client = get_clickhouse_client()

    # Insert cost metrics
    client.insert_cost_metrics([
        {
            'timestamp': datetime.now(),
            'customer_id': '123e4567-e89b-12d3-a456-426614174000',
            'cloud_provider': 'aws',
            'service_name': 'ec2',
            'instance_id': 'i-1234567',
            'instance_type': 'm5.xlarge',
            'region': 'us-east-1',
            'cost_per_hour': 0.192,
            'utilization_percent': 45.5,
            'is_spot': 0,
            'is_reserved': 0
        }
    ])

    # Query hourly costs
    results = client.query_cost_hourly(
        customer_id='123e4567-e89b-12d3-a456-426614174000',
        start_date=datetime.now() - timedelta(days=7),
        end_date=datetime.now()
    )
"""

from clickhouse_driver import Client
from typing import Dict, List, Any, Optional
import os
from datetime import datetime, timedelta
import logging

logger = logging.getLogger(__name__)
```

```python
class ClickHouseClient:
    """Client for inserting and querying time-series metrics in ClickHouse."""

    def __init__(self):
        """Initialize ClickHouse client with connection parameters."""
        self.client = Client(
            host=os.getenv('CLICKHOUSE_HOST', 'localhost'),
            port=int(os.getenv('CLICKHOUSE_PORT', 9000)),
            database=os.getenv('CLICKHOUSE_DB', 'optiinfra'),
            user=os.getenv('CLICKHOUSE_USER', 'default'),
            password=os.getenv('CLICKHOUSE_PASSWORD', '')
        )
        logger.info(f"ClickHouse client initialized: {self.client.connection.host}")

    def ping(self) -> bool:
        """
        Check if ClickHouse is accessible.

        Returns:
            bool: True if ClickHouse responds, False otherwise
        """
        try:
            result = self.client.execute('SELECT 1')
            return result[0][0] == 1
        except Exception as e:
            logger.error(f"ClickHouse ping failed: {e}")
            return False

    # ============================================================================
    # COST METRICS
    # ============================================================================

    def insert_cost_metrics(self, metrics: List[Dict[str, Any]]) -> int:
        """
        Insert cost metrics in batch.

        Args:
            metrics: List of cost metric dictionaries with keys:
                - timestamp (datetime)
                - customer_id (str UUID)
                - cloud_provider (str)
                - service_name (str)
                - instance_id (str)
                - instance_type (str)
```

```
            - region (str)
            - cost_per_hour (float)
            - utilization_percent (float)
            - is_spot (int: 0 or 1)
            - is_reserved (int: 0 or 1)

        Returns:
            int: Number of rows inserted

        Example:
            client.insert_cost_metrics([
                {
                    'timestamp': datetime.now(),
                    'customer_id': '123e4567-e89b-12d3-a456-426614174000',
                    'cloud_provider': 'aws',
                    'service_name': 'ec2',
                    'instance_id': 'i-1234567',
                    'instance_type': 'm5.xlarge',
                    'region': 'us-east-1',
                    'cost_per_hour': 0.192,
                    'utilization_percent': 45.5,
                    'is_spot': 0,
                    'is_reserved': 0
                }
            ])
        """
        if not metrics:
            return 0

        query = """
        INSERT INTO cost_metrics_ts
        (timestamp, customer_id, cloud_provider, service_name, instance_id,
         instance_type, region, cost_per_hour, utilization_percent, is_spot, is_reserved)
        VALUES
        """

        self.client.execute(query, metrics)
        logger.info(f"Inserted {len(metrics)} cost metrics")
        return len(metrics)

    def query_cost_hourly(
        self,
        customer_id: str,
        start_date: datetime,
```

```
    end_date: datetime,
    cloud_provider: Optional[str] = None
) -> List[Dict[str, Any]]:
    """
    Query hourly cost aggregations.

    Args:
        customer_id: Customer UUID
        start_date: Start datetime
        end_date: End datetime
        cloud_provider: Optional filter by cloud provider

    Returns:
        List of dictionaries with hourly cost data

    Example:
        results = client.query_cost_hourly(
            customer_id='123e4567-e89b-12d3-a456-426614174000',
            start_date=datetime.now() - timedelta(days=7),
            end_date=datetime.now(),
            cloud_provider='aws'
        )
    """
    query = """
    SELECT
        hour,
        cloud_provider,
        service_name,
        total_cost,
        avg_utilization,
        sample_count
    FROM cost_metrics_hourly_mv
    WHERE customer_id = %(customer_id)s
      AND hour >= %(start_date)s
      AND hour < %(end_date)s
    """

    params = {
        'customer_id': customer_id,
        'start_date': start_date,
        'end_date': end_date
    }

    if cloud_provider:
```

```
        query += " AND cloud_provider = %(cloud_provider)s"
        params['cloud_provider'] = cloud_provider

    query += " ORDER BY hour"

    result = self.client.execute(query, params)

    return [
        {
            'hour': row[0],
            'cloud_provider': row[1],
            'service_name': row[2],
            'total_cost': row[3],
            'avg_utilization': row[4],
            'sample_count': row[5]
        }
        for row in result
    ]

# ============================================================================
# PERFORMANCE METRICS
# ============================================================================

def insert_performance_metrics(self, metrics: List[Dict[str, Any]]) -> int:
    """
    Insert performance metrics in batch.

    Args:
        metrics: List of performance metric dictionaries

    Returns:
        int: Number of rows inserted
    """
    if not metrics:
        return 0

    query = """
    INSERT INTO performance_metrics_ts
    (timestamp, customer_id, service_id, service_type, model_name, request_id,
     latency_ms, throughput_tokens_per_sec, gpu_utilization, kv_cache_utilization,
     batch_size, prompt_tokens, completion_tokens, total_tokens)
    VALUES
    """
```

```
    self.client.execute(query, metrics)
    logger.info(f"Inserted {len(metrics)} performance metrics")
    return len(metrics)

def query_performance_p95(
    self,
    customer_id: str,
    service_id: str,
    hours: int = 24
) -> Dict[str, float]:
    """
    Query P95 latency over last N hours.

    Args:
        customer_id: Customer UUID
        service_id: Service UUID
        hours: Number of hours to query (default 24)

    Returns:
        Dictionary with performance metrics

    Example:
        results = client.query_performance_p95(
            customer_id='123e4567-e89b-12d3-a456-426614174000',
            service_id='456e7890-e89b-12d3-a456-426614174000',
            hours=24
        )
        # Returns: {'avg_latency_ms': 245.3, 'p95_latency_ms': 450.2, ...}
    """
    end_time = datetime.now()
    start_time = end_time - timedelta(hours=hours)

    query = """
    SELECT
        avgMerge(avg_latency) as avg_latency,
        quantileMerge(0.95)(p95_latency) as p95_latency,
        quantileMerge(0.99)(p99_latency) as p99_latency,
        avgMerge(avg_throughput) as avg_throughput,
        sum(request_count) as total_requests
    FROM performance_metrics_hourly_mv
    WHERE customer_id = %(customer_id)s
      AND service_id = %(service_id)s
      AND hour >= %(start_time)s
      AND hour < %(end_time)s
```

```
    """

    result = self.client.execute(query, {
        'customer_id': customer_id,
        'service_id': service_id,
        'start_time': start_time,
        'end_time': end_time
    })

    if not result:
        return {}

    row = result[0]
    return {
        'avg_latency_ms': row[0],
        'p95_latency_ms': row[1],
        'p99_latency_ms': row[2],
        'avg_throughput': row[3],
        'total_requests': row[4]
    }

# ============================================================================
# RESOURCE METRICS
# ============================================================================

def insert_resource_metrics(self, metrics: List[Dict[str, Any]]) -> int:
    """Insert resource metrics in batch."""
    if not metrics:
        return 0

    query = """
    INSERT INTO resource_metrics_ts
    (timestamp, customer_id, instance_id, instance_type, gpu_index,
     gpu_utilization, gpu_memory_used_mb, gpu_memory_total_mb, gpu_temperature,
     cpu_utilization, memory_used_gb, memory_total_gb, network_rx_mbps, network_tx_mbps)
    VALUES
    """

    self.client.execute(query, metrics)
    logger.info(f"Inserted {len(metrics)} resource metrics")
    return len(metrics)

def query_resource_utilization(
    self,
```

```
    customer_id: str,
    instance_id: str,
    hours: int = 24
) -> Dict[str, float]:
    """Query resource utilization over last N hours."""
    end_time = datetime.now()
    start_time = end_time - timedelta(hours=hours)

    query = """
    SELECT
        avgMerge(avg_gpu_util) as avg_gpu_util,
        maxMerge(max_gpu_util) as max_gpu_util,
        avgMerge(avg_gpu_memory) as avg_gpu_memory,
        avgMerge(avg_cpu_util) as avg_cpu_util
    FROM resource_metrics_hourly_mv
    WHERE customer_id = %(customer_id)s
      AND instance_id = %(instance_id)s
      AND hour >= %(start_time)s
      AND hour < %(end_time)s
    """

    result = self.client.execute(query, {
        'customer_id': customer_id,
        'instance_id': instance_id,
        'start_time': start_time,
        'end_time': end_time
    })

    if not result:
        return {}

    row = result[0]
    return {
        'avg_gpu_utilization': row[0],
        'max_gpu_utilization': row[1],
        'avg_gpu_memory_mb': row[2],
        'avg_cpu_utilization': row[3]
    }

# ============================================================================
# QUALITY METRICS
# ============================================================================

def insert_quality_metrics(self, metrics: List[Dict[str, Any]]) -> int:
```

```
    """Insert quality metrics in batch."""
    if not metrics:
        return 0

    query = """
    INSERT INTO quality_metrics_ts
    (timestamp, customer_id, service_id, request_id, model_name,
     relevance_score, coherence_score, factuality_score, hallucination_detected,
     toxicity_score, overall_quality_score, prompt_hash, latency_ms)
    VALUES
    """

    self.client.execute(query, metrics)
    logger.info(f"Inserted {len(metrics)} quality metrics")
    return len(metrics)

def query_quality_trends(
    self,
    customer_id: str,
    service_id: str,
    hours: int = 24
) -> List[Dict[str, Any]]:
    """Query quality trends over last N hours."""
    end_time = datetime.now()
    start_time = end_time - timedelta(hours=hours)

    query = """
    SELECT
        hour,
        avgMerge(avg_quality) as avg_quality,
        avgMerge(avg_relevance) as avg_relevance,
        avgMerge(avg_coherence) as avg_coherence,
        avgMerge(avg_factuality) as avg_factuality,
        sum(hallucination_count) as hallucinations,
        sum(request_count) as total_requests
    FROM quality_metrics_hourly_mv
    WHERE customer_id = %(customer_id)s
      AND service_id = %(service_id)s
      AND hour >= %(start_time)s
      AND hour < %(end_time)s
    GROUP BY hour
    ORDER BY hour
    """
```

```
        result = self.client.execute(query, {
            'customer_id': customer_id,
            'service_id': service_id,
            'start_time': start_time,
            'end_time': end_time
        })

        return [
            {
                'hour': row[0],
                'avg_quality': row[1],
                'avg_relevance': row[2],
                'avg_coherence': row[3],
                'avg_factuality': row[4],
                'hallucinations': row[5],
                'total_requests': row[6]
            }
            for row in result
        ]


# ============================================================================
# SINGLETON PATTERN
# ============================================================================

_clickhouse_client = None

def get_clickhouse_client() -> ClickHouseClient:
    """
    Get singleton ClickHouse client instance.

    Returns:
        ClickHouseClient: Singleton client instance

    Example:
        client = get_clickhouse_client()
        if client.ping():
            print("ClickHouse is ready!")
    """
    global _clickhouse_client
    if _clickhouse_client is None:
        _clickhouse_client = ClickHouseClient()
    return _clickhouse_client
```

📁 FILE 3: Package Initialization

Location: ~/optiinfra/shared/clickhouse/__init__.py

```
python

"""
ClickHouse time-series database package.

Provides high-performance storage and querying for:
- Cost metrics (hourly cloud spending)
- Performance metrics (per-request LLM latency)
- Resource metrics (GPU/CPU utilization)
- Quality metrics (LLM output quality scores)

Usage:
    from shared.clickhouse import get_clickhouse_client

    client = get_clickhouse_client()

    # Insert metrics
    client.insert_cost_metrics([...])

    # Query aggregations
    results = client.query_cost_hourly(...)
"""

from shared.clickhouse.client import (
    ClickHouseClient,
    get_clickhouse_client
)

__all__ = [
    'ClickHouseClient',
    'get_clickhouse_client'
]
```

📁 FILE 4: Schema Initialization Helper

Location: ~/optiinfra/shared/clickhouse/schemas/__init__.py

```python
"""
ClickHouse schema initialization helpers.
"""

# Empty file for now, but can add schema utilities later
```

📁 FILE 5: Update Requirements

Location: `~/optiinfra/shared/requirements.txt`

```txt
# Existing dependencies...
sqlalchemy==2.0.23
alembic==1.12.1
psycopg2-binary==2.9.9

# ClickHouse driver (ADD THIS)
clickhouse-driver==0.2.6

# Other dependencies...
```

📁 FILE 6: README Documentation

Location: `~/optiinfra/shared/clickhouse/README.md`

```markdown
```

ClickHouse Time-Series Database

High-performance time-series storage for OptiInfra metrics.

Overview

ClickHouse stores high-frequency metrics that would overwhelm PostgreSQL:
- **1M+ inserts/second** vs PostgreSQL's ~10K/second
- **10-20x compression** vs PostgreSQL's 2-3x
- **Millisecond queries** vs PostgreSQL's seconds

Architecture

Tables
1. **cost_metrics_ts** - Cloud costs (1-hour granularity, 90-day retention)
2. **performance_metrics_ts** - LLM latency (per-request, 30-day retention)
3. **resource_metrics_ts** - GPU/CPU utilization (1-minute, 90-day retention)
4. **quality_metrics_ts** - Quality scores (per-request, 30-day retention)

Materialized Views
Pre-aggregated hourly rollups for fast dashboard queries:
- cost_metrics_hourly_mv
- performance_metrics_hourly_mv
- resource_metrics_hourly_mv
- quality_metrics_hourly_mv

Usage

Initialize Database
```bash
# Run initialization script
docker exec -i optiinfra-clickhouse clickhouse-client < shared/clickhouse/migrations/init.sql
```

Python Client
```python
from shared.clickhouse import get_clickhouse_client
from datetime import datetime, timedelta

# Get client
client = get_clickhouse_client()

# Check connection
```

```
if client.ping():
    print("✅ ClickHouse connected!")

# Insert
```